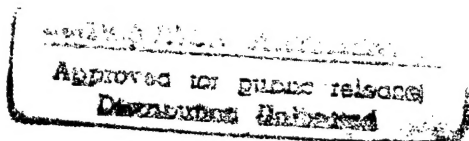# Supervisor Calls

Tera Computer Company
2815 Eastlake Ave E
Seattle, WA 98102

September 9, 1992

1 Introduction

This document provides a high level overview of how supervisor calls are implemented. For implementation details, the reader should refer to the implementation files (svcEntry.w, svcTable.w, svcParam.w, svcCallUser.w).

2 Overview

Access to operating system services is provided by a collection of supervisor libraries. The libraries are permanently resident in the program memories of all processors, and are mapped into the absolute pages with supervisor level privileges. Thus they appear transparently in the address space of all executing tasks.

To execute a supervisor call, a user stream simply branches to the entry point of the appropriate supervisor library routine. The change of privilege level occurs as a side-effect of subroutine invocation. The entry point for each supervisor routine contains a level_enter instruction, which allows the stream to temporarily acquire supervisor privileges. Likewise, the supervisor routine exit point contains a level_return instruction, which returns the privilege level of the stream to user mode. Thus, the entry and exit points serve as gateways between the user and supervisor privilege levels.

3 Establishing Addressability

The Tera compiler generates position independent code. Each function has a linkage section associated with it, which is stored in a linkage segment in the task data address space. The linkage section contains various information, including the text address of the function, and addresses of external references. When calling a function, the caller supplies the linkage pointer for the function in a known register. This convention does not work for supervisor libraries, because the user-level code cannot be trusted to provide the correct linkage section pointer to the supervisor. Therefore a mechanism for establishing addressability for privileged routines must be provided.

A straightforward solution is to provide a system table which contains the linkage pointers and perform a table lookup on entry to the supervisor. Currently, the table is defined statically, and the supervisor call index (SVC number) is hard coded in a corresponding user-level stub routine.

The index is provided as a parameter to a generic supervisor level stub routine, which performs a table lookup to retrieve the linkage pointer and calls the actual supervisor routine.

Figure **??** illustrates what happens when a user calls a supervisor privileged library routine, sysFunc. The entry point for sysFunc is a user library routine, which packages the user arguments together with the SVC number and calls a generic supervisor level entry routine. The first instruction in the supervisor stub is a `level_enter` instruction, which promotes the privilege level. It initializes a supervisor environment. It then looks up the linkage pointer for the actual supervisor routine and calls the routine. The actual routine is written as a normal C routine. It returns to the generic stub, which restores the user environment and does a `level_return` to the user stub.

The supervisor entry routine follows the regular compiler calling conventions for "C" code. The amount of user state saved and restored by the supervisor entry routine is minimal, consisting of the user CCB, stack pointer, and trap handler. The entry routine uses only "caller-save" registers for scratch; the "callee-save" registers are saved and restored by the actual supervisor routine, if it uses them. The amount of state needed to initialize the supervisor environment is also minimal, consisting of a CCB, stack pointer, end of stack pointer, and trap handler.

A static table is simple and fast. However, the need to recompile the user stub library whenever the table contents change is a drawback. The goal is to eventually convert to a dynamic table, which is more flexible, but also more complex to implement. In this approach, the table is dynamically generated by the linker/loader as the supervisor libraries are loaded. A special *build tool* is used to register the supervisor routine. Registration causes an entry to be allocated in the table which is initialized to contain the linkage pointer to the routine. It also generates a small supervisor level stub which does the table lookup and passes the linkage pointer to a generic init routine. The init routine initializes the supervisor environment and jumps to the supervisor routine. The return
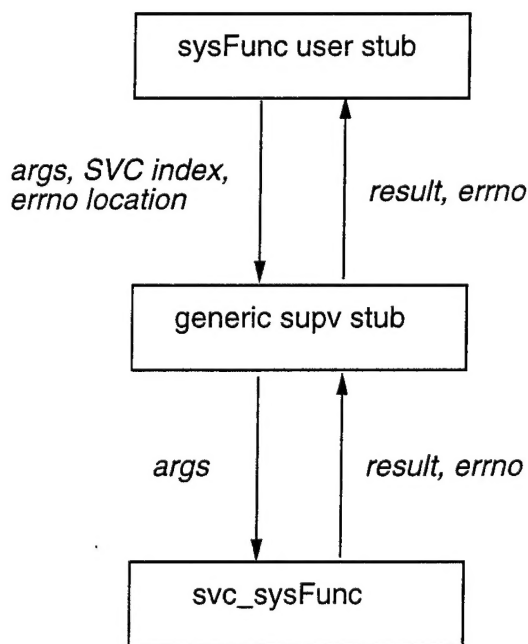

FIGURE 1: Supervisor Call Control Flow

address is fudged so that it returns directly from the init routine to user code. The addresses of the stub routines are exported as a shared library which the user program links against at load time.

4   User-level Entry Stub

This section describes the user-level stub routine. The stub routine is responsible for processing error codes, and for providing information about the routine to a generic supervisor level entry stub. The user stubs are written as normal C functions, and are collected into a shared library which the user program links against at load time (note: the program may be statically linked, but the version numbers must be checked at load time to ensure consistency). The pseudocode below is an example of the user stub for the mythical supervisor call sysFunc.

```
<user stub pseudocode>≡
int sysFunc(arg1, arg2) {
    int result;
    RT_CCB * spare_ccb;
    spare_ccb = rt_alloc_ccb();
    ccb->errno = spare_ccb;                                                     5
    do {
        result = svc_entry_2b(arg1, arg2, SVC_SYSFUNC, &ccb->errno);
    } while (ccb->errno == SV_MAX);
    if (ccb->errno != 0) result = -1;
    rt_free_ccb(spare_ccb);                                                    10
    return result;
}
```

Some supervisor calls may suspend pending the completion of long term events. When suspension occurs, the user level runtime must be notified via an upcall. In order for the upcall to execute, a user chore control block (ccb) is required. The user stub calls a runtime routine to allocate the spare ccb and passes it as an argument to the supervisor entry routine (to minimize the number of arguments passed, the spare ccb is passed in the error code location). The spare ccb is deallocated upon return from the supervisor. (Note: the spare ccb allocation and deallocation calls are omitted for non-blocking supervisor calls.)

The system provides a small set of supervisor entry stub routines which are customized according to the number of parameters, and whether the call is allowed to block. The example presented here requires two parameters (arg1, arg2), and is a blocking call. The supervisor call number (SVC_SYSFUNC, currently hard coded) is also passed to the entry routine, which performs a table lookup to locate the linkage pointer for the actual supervisor routine. The user stub also provides a location (ccb->errno) to contain the error code from the supervisor call.

The values returned by the user stub follow the Unix convention. The return value is the value returned by the actual supervisor call if the call completed successfully. Otherwise, the errno field in the user chore control block contains the error code, and the value -1 is returned. (Notes: Eventually the Tera compiler will support structure return values passed in registers, in which case the error code parameter will no longer be needed.)

System resources are reserved while upcalls are in progress. Thus, in order to protect itself against malicious or errant users, the system restricts the total number of concurrent upcalls in progress on a per-team basis. In practice, this translates to restricting the number of system calls in progress. When the limit is exceeded, the supervisor call fails and an error code is returned (SVC_MAX). In order to insulate the user code from having to deal with this Tera specific error code, the user stub examines the error code and reissues the supervisor call. The limit is normally set to a value which is large enough to ensure that failure is an extremely rare event.

5 Generic Supervisor Entry Routine

This section describes the supervisor entry stub routine. The entry points for the supervisor stub routines are exported symbols by the operating system. For the user program, they are imported symbols which are resolved at load time by the dynamic linker/loader. At boot time, the normal shared library mechanism is used to load exported supervisor symbols into system defined tables (not implemented yet).

The entry routines must be written in assembler, because direct access to the registers is required. There are a small number of supervisor entry routines, which provide customized initialization sequences based on the characteristics of the supervisor routines they service (for instance, the number of parameters passed, or whether the call is allowed to block). Lightweight supervisor routines can be implemented entirely within a custom stub routine (for instance, a routine which returns the current protection domain number); such routines may not even require a supervisor stack or trap handler.

The entry routine expects two arguments (SVC index and errno address) in addition to the arguments for the actual supervisor routine. In order to avoid having to repack the user arguments, the index and errno address should occupy the last two positions in the argument list. For supervisor calls with six or less parameters, all arguments are passed in registers. For supervisor calls which require more than six parameters, the supervisor stub must retrieve its arguments from the argument block provided by the compiler. For calls with more than eight arguments, the supervisor stub must copy the user argument block into supervisor space prior to calling the actual supervisor routine. (Note: currently there are no Unix system calls which require more than six arguments.)

The entry routine is responsible for installing the privileged environment on entry to the supervisor. The environment is minimal, and consists of a chore control block (which contains the initial stack) and a trap handler. To enable fast allocation of supervisor chore control blocks, the system maintains a pool of preinitialized blocks on a per-team basis. For safety, lookahead is prohibited and all traps are required to be disabled upon entry to the supervisor. This isolates traps generated from within the user, preventing the user-level trap handler from being inadvertently invoked while in privileged code. The supervisor entry routine saves the user trap handler, stack, and chore control block pointers and replaces them with supervisor privileged versions. The user versions are restored on exit from the supervisor.

6 Parameter Validation

Supervisor calls cannot trust addresses supplied by the user as parameters. The hardware provides a special instruction to validate user addresses: the probe instruction checks the validity of the

address and returns a pointer to the last byte in the segment. However, simply doing a probe prior to access is not sufficient, since another user stream could remove or replace the memory mapping prior to the memory access. There are two cases which need to be considered: large transfers involving IOPs, and small transfers.

Supervisor calls which need to transfer large amounts of data to or from the user address space use IOPs in loopback mode to do the copy. Since the IOP is passed a physical address (in the form of a private data map entry), the relevant physical data segments must be wired down during the transfer. This can be done by setting a flag in the data segment descriptor which indicates that a transfer is in progress. It is not necessary to lock the virtual memory map entry. If the entry is changed during the transfer, the user may lose access to the data, but the transfer may safely complete.

For transfers involving a small number of words (for instance, to fill in a structure passed by reference), we can avoid the overhead of looking up and locking the virtual address map entry, if we are willing to handle any resulting traps caused by the memory accesses. This is an optimistic approach, which is based on the assumption that the vast majority of memory accesses will succeed. Provided the system does not allow an existing user-level mapping to be replaced by a privileged mapping, it is safe to probe the address range to validate privileges, then proceed with the memory access (i.e. there is no danger of reading or writing privileged memory). Any memory trap which occurs is handled by the supervisor as a bad user address, and causes the supervisor call to abort.

A set of validation functions are provided, which are used to check the validity of user address ranges (`validateUserAny`, `validateUserRead`, `validateUserWrite`, `validateUserModify`). A pair of routines (`copyin`, `copyout`) are also provided for transferring data between the user and supervisor privilege levels. These routines are written in assembler, and perform word or byte-aligned copies, as appropriate. The appropriate validation routine should be called once prior to using the copy routines to access the address range.

7 Upcalls

When a supervisor-promoted chore suspends waiting on an operating system condition, the user runtime is notified via an upcall. The runtime responds by blocking the user chore which made the supervisor call. When the operating system condition is satisfied, a second upcall is made to the user runtime requesting resumption of the blocked chore. The runtime responds by resuming the blocked chore, which executes a special supervisor call which resumes the suspended supervisor chore.

From the perspective of the supervisor, user upcalls cannot be trusted. The supervisor defends itself against possible user errors and makes no assumptions about the timely return of a supervisor chore following an upcall. Note that restarting a supervisor after a timeout delay is problematic. For one, it is not generally possible to distinguish normal delays (due to trap processing, for instance) from delays caused by user-level errors, so choosing an appropriate timeout value is difficult. Second, following a timeout, the supervisor cannot simply abort the system call and return an error code to the caller, since the user chore may have already been blocked and its virtual processor reassigned.

Instead, the system restricts the total number of concurrent upcalls in progress on a per-team basis. In practice, this translates to restricting the number of system calls in progress. If the maximum is reached, any further system calls will fail. At task termination time, any supervisors which are blocked on an upcall are reclaimed. Thus, it is imperative that a supervisor not hold any system

locks when the upcall is made. In fact, the supervisor must be expendable; i.e. there should be no system dependence on the proper resumption of a supervisor following an upcall.

The correct operation of the upcall/callback sequence is dependent on the user executing the correct sequence of calls with the correct parameters. Incorrect interleaving of calls or mangled parameters could result in havoc. However, user errors can only affect supervisors executing within the user task. They cannot affect the operation of the rest of the system. The callback routines attempt to catch blatant errors. One possibility for handling these error conditions is to send a signal to the user which causes it to terminate execution.

8   Crossing the Supervisor-Kernel Boundary

Supervisor chores access kernel services by invoking kernel library routines. The kernel library is similar to the supervisor libraries in that all entry points are protected by a `level_enter` instruction, which allows the stream to temporarily acquire kernel privileges. Some consideration was given to the tradeoffs between protection and performance in determining the amount of supervisor state that must be saved, replaced, and restored across kernel calls. The conclusion reached was that both performance and reliability are important considerations. A firewall between the supervisor and kernel would greatly aid OS debugging. However, the firewall should be designed in such way that it can be selectively compiled out. Subsequently, as indicated by performance test results, the firewall should be reduced or removed for production systems. There is particular concern that placing the KIPC system in the kernel could have a negative impact on KIPC performance in the supervisor.

The supervisor state replaced by kernel state consists of the trap handler, CCB, trap save area, and stack segment. Since the supervisor is trusted not to be malicious, it is safe to trust the linkage pointer provided by the supervisor for the kernel call. Thus, there is no need for a table lookup. The code to save and restore the supervisor environment, and to allocate and free the kernel environment is encapsulated in a generic kernel entry routine. The approach is similar to that used for supervisor calls. The kernel entry stub serves as a wrapper around a call to the actual kernel routine. Note that the general registers do not need to be saved in the kernel entry stub, since thay are saved and restored according to normal compiler conventions by the actual kernel routine.

In a production system, the protection bits can be used to protect kernel data structures from the supervisor. This is accomplished by typing kernel data structures with the *protected* key word. This sets the datatrap1 bit in the memory locations containing those data structures. Code which accesses protected data structures is compiled with a special flag, which causes all accesses to protected memory locations to have datatrap1 disabled. A vanilla access to a protected data structure results in a datatrap1 exception. The only caveat is that the use of datatrap1 for protection conflicts with its use for setting breakpoints by the debugger. Thus, the protection bits should be enabled only for a fully debugged kernel.